Universidad Nacional de Córdoba
Facultad de Ciencias Exactas, Físicas y Naturales
Escuela de Ingeniería en Computación

Trabajo Práctico 1

Programación Concurrente

Integrantes: Cabrera, Augusto Gabriel
Mansilla, Josías Leonel
Moroz, Esteban Mauricio
Pallardó, Agustín
Villar, Federico Ignacio
Profesores: Ludemann, Mauricio
Ventre, Luis Orlando

Fecha de entrega: 25 de abril
Córdoba

Resumen

En el siguiente informe se adjunta todo el procedimiento de trabajo implementado para la realización del primer trabajo práctico de la materia Programación Concurrente.

En un inicio, se plantea el diagrama de clases del programa a implementar, para luego, esbozar la estructura básica del mismo implementando lo solicitado por el enunciado. Una vez obtenido un programa funcional, se procedió a realizar diferentes pruebas para corroborar el correcto funcionamiento del mismo y encontrar posibles falencias, entre las que podrían haber problemas de concurrencia.

Finalmente, se analizaron los resultados obtenidos y se elaboraron conclusiones en base a los mismos. En este informe se encuentran además, diferentes explicaciones que contrinuyen al correcto entendimiento del programa desarrollado, así como a la comprobación del correcto funcionamiento del mismo.

Índice de Contenidos

Índice de Figuras

Índice de Códigos

1. Consigna

1.1. Enunciado

En un sistema de gestión de imágenes, existe una funcionalidad que se encarga de mejorar la calidad de los archivos obtenidos, previo a la entrega final al cliente. Esta funcionalidad ha quedad obsoleta, por lo tanto se solicita realizarlo nuevamente, teniendo en cuenta los siguientes lineamientos de diseño.

Dicha funcionalidad se divide en **cuatro procesos**.

El **primer proceso** se encarga de cargar las imágenes en un contenedor, en su estado original. Existen **dos hilos** que ejecutan este proceso, demorando un tiempo aleatorio en ms para obtener la imagen y depositarla en el contenedor. Ambos hilos dejan las imágenes en el mismo contenedor.

El **segundo proceso** tiene por objetivo mejorar la iluminación de cada imagen. En este caso, **tres hilos** ejecutan este proceso, demorando un **tiempo aleatorio** en ms para mejorar la iluminación de cada imagen. Cada hilo debe tomar una **imagen aleatoria** del contenedor por vez, y no puede tomar más de una vez la misma imagen. En el tiempo que demora en mejorar la imagen no debe bloquear otros hilos que quieran tomar otras imágenes para mejorar. Cada imagen debe ser mejorada por **todos los hilos** (los 3 del presente proceso) para que el siguiente proceso pueda tomarla.

El **tercer proceso** debe ajustar las imágenes al tamaño final solicitado. Este proceso tiene que ejecutarse después que la imagen ya fue mejorada. **Tres hilos** son los encargados de ejecutar este proceso, tomando cada uno una **imagen aleatoria** del contenedor por vez para ajustarla, y demorando un **tiempo aleatorio** en ms. En el tiempo que demora en ajustar la imagen no debe bloquear otros hilos que quieran tomar otras imágenes para ajustar. Una imagen solo puede ser ajustada una sola vez.

El **cuarto proceso** toma las imágenes ya mejoradas y ajustadas, les hace una copia en el contenedor final, y luego las elimina del contenedor final, y luego las elimina del contenedor inicial. **Dos hilos** ejecutan este proceso, y demoran un tiempo en ms en realizar su trabajo. Cada hilo toma de a una imagen por vez de **manera aleatoria**.

1.1.1. Consideraciones

- Los objetos tienen un atributo ïmprovements", el cual registra cuántos hilos han tomado el archivo para mejorarlo (proceso 2).
- Cuando se menciona un tiempo aleatorio en ms, el mismo debe ser no nulo y a elección del grupo.
- Una imagen no puede pasar a ser ajustada sin antes haber sido mejorada por los 3 hilos del segundo proceso.
- Al finalizar la ejecución es necesario verificar cuántas imágenes movió del contenedor inicial hacia el contenedor final, cada hilo del cuarto proceso.
- El sistema debe contar con un LOG con fines estadísticos, el cual registre cada 500 ms en un archivo:
  - Cantidad de imágenes insertadas en el contenedor.
  - Cantidad de imágenes mejoradas complementamente.

- Cantidad imágenes ajustadas.
- Cantidad de imágenes que han finalizado el último proceso.
- El estado de cada hilo del sistema.

1.1.2. Ejercicios

1. Hacer un diagrama de clases que modele el sistema de datos con TODOS los actores y partes.
2. Hacer un solo diagrama de secuencias que muestre la siguiente interacción:
   A Con el contenedor inicial vacío, la inserción de un archivo.
   B La interacción de un hilo del segundo proceso al intentar acceder al archivo para la mejora.
   C La interacción de un hilo del tercer proceso cuando accede a un archivo para ajustar el tamaño de la imagen.
3. Las demoras del sistema en sus cuatro procesos deben configurarse de tal manera de poder procesar 100 imágenes (desde la inserción en el primer contender hasta que son movidas al contenedor final) en un periodo mínimo de 10 segundos y máximo de 20 segundos.
4. Hacer un análisis detallado de los tiempos que el programa demora. Luego contrastarlo con múltiples ejecuciones, obteniendo las conclusiones pertinentes.
5. El grupo debe poder explicar los motivos de los resultados obtenidos. Y los tiempos del sistema.
6. Debe haber una clase Main que al correrla, inicie los hilos.

2. Código de Java

2.1. Funcionamiento

Se crearon las instancias de las clases y variables que se utilizarán. Luego, se inicializan los arreglos de los distintos actores. Se lo hizo de la siguiente forma: 2 hilos **Loader**, 3 hilos **Improver**, 3 hilos **Resizer** y 2 hilos **Cloner**. Además se crea un objeto de la clase **Log**. Posteriormente, se inicia el funcionamiento de todos los hilos y comienza la ejecución del programa.

2.1.1. Actores

- **Loader**: se encarga de crear instancias de la clase **Image** y guardarlas dentro de **initContainer**. Esta acción se repite de manera iterada mientras que la cantidad total de datos agregados al **initContainer** sea menor a la establecida previamente como objetivo (**targetAmountOfData**). El **Loader** posee un contador que lleva el registro de la cantidad total de imágenes que ha agregado al **initContainer** y se puede así consultar su valor a través del método **getImageLoad()**. Si el **initContainer** está lleno, se lanza una excepción para evitar que otro hilo entre al bloque **synchronized** del método **Load()** y no agregue otra imagen al contenedor inicial. Finalmente, el hilo duerme 50 milisegundos entre iteraciones.

- **Improver**: se encarga de chequear por única vez cada instancia de la clase **Imagen** que está almacenada en el **initContainer**. Esto se va a repetir mientras que el proceso de creación de datos no haya sido finalizado y el **initContainer** contenga elementos. Los hilos **Improvers** chequean que la imagen tomada con el método **isImproved()**. Si esta última no fue modificada por el **Improver** correspondiente, lo agrega a la lista enalazada **Improvements** para que no se vuelva a modificar (ser tomada) por el mismo hilo dos veces. Una vez que los tres hilos mejoren la imagen, el proceso **improve()** cambia el valor del campo **iAmImproved** a **true**, indicando que la imagen en cuestión ya fue mejorada tres veces, como se requería, garantizando que fue mejorada una vez por cada uno de los hilos. Finalmente, el hilo duerme por 100 milisegundos entre iteraciones.

- **Resizer**: se encarga de tomar una imagen aleatoria del contenedor inicial y corroborar que fue mejorada por los tres **Improver** haciendo uso del método getter llamado **getIamImproved()**. Luego, una vez encontrada una imagen que cumple con el requisito, se procede a realizarle el **resize()**, que cambia el valor del campo **resized** a **true**, indicando así que la imagen ya fue reescalada y aumenta el valor del contador **totalImagesResized** con el método **increaseImageResizer()**. Por último, realiza un sleep de 100 milisegundos.

- **Cloner**: se encarga de tomar una imagen aleatoria del contenedor inicial, corroborar que fue reescalada, y, si lo fue, mediante el proceso **copyAndDelete()** la toma y la copia en el contenedor final, borrándola del contenedor inicial. Para ello, se hace uso del método **tryCloneToFinalContainer()** para evitar problemas de concurrencia y comprobar si ya fue agregada al contenedor final. Esta comprobación se hace porque si dos hilos ingresan al método **clone()** a la vez, uno de los dos cambia el valor de la variable **increaseImageClone()**, esto es para poder mostrar por **Log** la cantidad de imágenes clonadas por cada hilo. Finalmente se duerme por 50 milisegundos.

- **Log**: se encarga de imprimir en un archivo de texto llamado **Estadistica.txt** la información acerca del estado del programa en intervalos de tiempo de 500 milisegundos.

2.1.2. Implementación

El programa desarrollado cuenta con las siguientes clases:

- **Cloner**
- **Container**
- **FinalContainer**
- **Image**
- **Improver**
- **InitConainer**
- **Loader**
- **Log**
- **Main**
- **Resizer**

Se implementan de la siguiente forma:

Código 1: Clase Main

```
1  package TP1;
2
3  public class Main {
4    private static final int numberOfLoaders = 2;
5    private static final int numberOfImprovers = 3;
6    private static final int numberOfResizers = 3;
7    private static final int numberOfClones = 2;
8    private static final int targetAmountOfData = 100;
9
10   public static void main(String[] args) {
11     long start = System.currentTimeMillis();
12     Log.clearFile();
13
14     InitContainer initContainer = new InitContainer(targetAmountOfData);
15     FinalContainer finalContainer = new FinalContainer(targetAmountOfData);
16
17     Loader[] loaders = new Loader[numberOfLoaders];
18     Cloner[] cloners = new Cloner[numberOfClones];
19     Resizer[] resizers = new Resizer[numberOfResizers];
20     Improver[] improvers = new Improver[numberOfImprovers];
21
22     Thread[] loadersThreads = new Thread[numberOfLoaders];
23     Thread[] improversThreads = new Thread[numberOfImprovers];
24     Thread[] resizerThreads = new Thread[numberOfResizers];
25     Thread[] clonersThreads = new Thread[numberOfClones];
26
```

```
27          TP1.Log log = new TP1.Log(targetAmountOfData, initContainer, finalContainer, loaders,
    ↪ improvers, resizers,
28                  cloners, loadersThreads, improversThreads, resizerThreads, clonersThreads);
29
30          for (int i = 0; i < numberOfLoaders; i++) {
31            loaders[i] = new Loader(initContainer, "TP1.Loader " + i);
32            loadersThreads[i] = new Thread(loaders[i]);
33            loadersThreads[i].setName(loaders[i].getName() + " (Thread ID: " +
    ↪ loadersThreads[i].getId() + ")");
34          }
35
36          for (int i = 0; i < numberOfImprovers; i++) {
37            improvers[i] = new Improver(initContainer, "TP1.Improver " + i, numberOfImprovers);
38            improversThreads[i] = new Thread(improvers[i]);
39            improversThreads[i].setName(improvers[i].getName() + " (Thread ID: " +
    ↪ improversThreads[i].getId() + ")");
40          }
41
42          for (int i = 0; i < numberOfResizers; i++) {
43            resizers[i] = new Resizer(initContainer, "TP1.Resizer " + i);
44            resizerThreads[i] = new Thread(resizers[i]);
45            resizerThreads[i].setName(resizers[i].getName() + " (Thread ID: " +
    ↪ resizerThreads[i].getId() + ")");
46          }
47
48          for (int i = 0; i < numberOfClones; i++) {
49            cloners[i] = new Cloner(initContainer, finalContainer, "TP1.Cloner " + i);
50            clonersThreads[i] = new Thread(cloners[i]);
51            clonersThreads[i].setName(cloners[i].getName() + " (Thread ID: " +
    ↪ clonersThreads[i].getId() + ")");
52          }
53
54          for (Thread loadersThread : loadersThreads) {
55            loadersThread.start();
56          }
57          for (Thread improverThread : improversThreads) {
58            improverThread.start();
59          }
60          for (Thread resizersThread : resizerThreads) {
61            resizersThread.start();
62          }
63          for (Thread clonerThread : clonersThreads) {
64            clonerThread.start();
65          }
66          log.start();
67
68          try {
69            for (Thread waiting : loadersThreads) {
70              waiting.join();
71            }
72            for (Thread waiting : improversThreads) {
```

```
73              waiting.join();
74            }
75            for (Thread waiting : resizerThreads) {
76              waiting.join();
77            }
78            for (Thread waiting : clonersThreads) {
79              waiting.join();
80            }
81          } catch (InterruptedException e) {
82            e.printStackTrace();
83          }
84          log.interrupt();
85          long finish = System.currentTimeMillis();
86          long timeElapsed = finish - start;
87          System.out.println("Elapsed Time: " + (float) (timeElapsed / 1000.00) + " seconds");
88        }
89      }
90
```

Código 2: Clase Cloner

```
1   package TP1;
2
3   import java.util.concurrent.TimeUnit;
4
5   public class Cloner implements Runnable {
6
7     private final InitContainer initContainer;
8
9     private final FinalContainer finalContainer;
10
11    private final String name;
12
13    private Image lastImageClone;
14
15    private int imageCloned;
16
17    public Cloner(InitContainer initContainer, FinalContainer finalContainer, String name) {
18
19      this.initContainer = initContainer;
20      this.finalContainer = finalContainer;
21      this.name = name;
22      lastImageClone = null;
23      imageCloned = 0;
24
25    }
26
27    @Override
28    public void run() {
29      while (initContainer.getSize() > 0 || initContainer.isNotLoadCompleted()) {
30        try {
```

```
31              lastImageClone = initContainer.getImage(lastImageClone);
32              if (lastImageClone != null) {
33
34                if (lastImageClone.isResized()) {
35                  if (!lastImageClone.isIamDeletefromInitContainer()) {
36                    if (lastImageClone.tryCloneToFinalContainer()) {
37                      if (finalContainer.Clone(initContainer.CopyAndDeleted(lastImageClone), this,
38                          imageCloned)) {
39                        increaseImageClone();
40                        TimeUnit.MILLISECONDS.sleep(50);
41                      }
42
43                    }
44                  }
45                }
46              }
47            } catch (InterruptedException | IllegalArgumentException e) {
48              e.printStackTrace();
49              break;
50
51            } catch (NullPointerException | IndexOutOfBoundsException e) {
52              e.printStackTrace();
53            }
54          }
55        }
56
57        public String getName() {
58          return name;
59        }
60
61        public int getImageCloned() {
62          return imageCloned;
63        }
64
65        public void increaseImageClone() {
66          imageCloned++;
67        }
68
69      }
70
```

Código 3: Clase Container

```
1     package TP1;
2
3     import java.util.LinkedList;
4
5     public abstract class Container {
6       protected LinkedList<Image> container;
7       protected int targetAmountOfImages;
8
```

```
9       public Container() {
10        container = new LinkedList<>();
11      }
12
13      public int getSize() {
14        return container.size();
15      }
16    }
17
```

Código 4: Clase FinalContainer

```
1   package TP1;
2
3   public class FinalContainer extends Container {
4
5     private boolean cloneCompleted;
6     private int targetAmountOfImages; // cantidad Max de imagenes a trabajar
7     private int amountOfImages; // cantidad de imagenes actual
8
9     public FinalContainer(int targetAmountOfImages) {
10      this.amountOfImages = 0;
11      this.targetAmountOfImages = targetAmountOfImages;
12      cloneCompleted = false;
13
14    }
15
16    public synchronized boolean Clone(Image image, Cloner cloner, int cantidad) throws
    ↪ InterruptedException {
17      try {
18        if (!cloneCompleted && image != null && image.isResized()) { // agregamos esto pq el
    ↪ return null de
19                                                  // DopyandDelete nos perjudicaba
20          this.container.addLast(image);
21          amountOfImages++;
22          System.out.printf("[FinalContainer (Size: %d)] %s Image cloned <ID: %d \n",
23              this.container.size(),
24              Thread.currentThread().getName(), image.getId());
25          if (amountOfImages == targetAmountOfImages) {
26            cloneCompleted = true;
27            cloner.increaseImageClone();
28            throw new InterruptedException("Contenedor final lleno");
29          }
30        }
31      } catch (NullPointerException e) {
32        System.out.println("Imagen a clonar nula");
33      }
34      return !cloneCompleted;
35    }
36  }
37
```

Código 5: Clase Image

```
1  package TP1;
2
3  import java.util.ArrayList;
4  import java.util.List;
5
6  public class Image {
7
8    private final List<Improver> improvements;
9    private final int id;
10   private static int generator = 0;
11   private static final Object key = new Object();
12   private boolean resized;
13   private boolean clonedToFinalContainer;
14
15   // Defino las llaves
16
17   private static final Object keyImprove = new Object();
18   private static final Object keyResize = new Object();
19   private static final Object keyClone = new Object();
20
21   private boolean IamDeletefromInitContainer;
22
23   private boolean iamImproved;
24
25   private static int newId() {
26     synchronized (key) {
27       return generator++;
28     }
29   }
30
31   public Image() {
32     improvements = new ArrayList<>();
33     resized = false;
34     id = newId();
35     clonedToFinalContainer = false;
36     iamImproved = false;
37     IamDeletefromInitContainer = false;
38   }
39
40   public List<Improver> getImprovements() {
41     return improvements;
42   }
43
44   public static int getGenerator() {
45     return generator;
46   }
47
```

```
48      public static void setGenerator(int generator) {
49        Image.generator = generator;
50      }
51
52      public void setResized(boolean resized) {
53        this.resized = resized;
54      }
55
56      public void setClonedToFinalContainer(boolean clonedToFinalContainer) {
57        this.clonedToFinalContainer = clonedToFinalContainer;
58      }
59
60      public boolean isIamImproved() {
61        return iamImproved;
62      }
63
64      public void setIamImproved(boolean iamImproved) {
65        this.iamImproved = iamImproved;
66      }
67
68      public Image(List<Improver> improvements, boolean resized, int id, boolean
    ↪ clonedToFinalContainer,
69          boolean iamImproved) { // es solamente para el Clone
70        this.improvements = improvements;
71        this.resized = resized;
72        this.id = id;
73        this.clonedToFinalContainer = clonedToFinalContainer;
74        this.iamImproved = iamImproved;
75      }
76
77      public boolean isIamDeletefromInitContainer() {
78        return IamDeletefromInitContainer;
79      }
80
81      public boolean improve(Improver improver) {
82
83        synchronized (keyImprove) {
84          improvements.add(improver);
85          if (improvements.size() == improver.getTotalThreadsImprovements()) {
86            this.setIamImprove();
87            return true;
88          } else {
89            return false;
90          }
91        }
92      }
93
94      public boolean isImproved(Improver improver) {
95        return improvements.contains(improver);
96      }
97
```

```
98        public boolean resize() {
99          synchronized (keyResize) {
100           if (!isResized()) {
101             resized = true;
102             return true;
103           } else {
104             return false;
105           }
106         }
107       }
108
109       public int getId() {
110         return id;
111       }
112
113       public void setIamImprove() {
114         iamImproved = true;
115       }
116
117       public boolean isResized() {
118         return resized;
119       }
120
121       public boolean tryCloneToFinalContainer() {
122         synchronized (keyClone) {
123           if (!IamDeletefromInitContainer) {
124             IamDeletefromInitContainer = true;
125             return true;
126           } else {
127             return false;
128           }
129         }
130       }
131
132       public boolean getAmIImproved() {
133         return iamImproved;
134       }
135
136       public boolean getAmIResized() {
137         return resized;
138       }
139     }
140
```

Código 6: Clase Improver

```
1   package TP1;
2
3   import java.util.concurrent.TimeUnit;
4
5   public class Improver implements Runnable {
```

```
6
7       private final InitContainer initContainer;
8
9       private int totalImprovements;
10
11      private int totalImagesImprovedByThread;
12
13      private final String name;
14
15      private final int totalThreadsImprovements;
16
17      private Image lastImageImprove;
18
19      public Improver(InitContainer initContainer, String name, int totalThreadsImprovements) {
20        this.initContainer = initContainer;
21        totalImprovements = 0;
22        this.name = name;
23        this.totalThreadsImprovements = totalThreadsImprovements;
24        totalImagesImprovedByThread = 0;
25        lastImageImprove = null;
26      }
27
28      @Override
29      public void run() {
30        while (initContainer.getSize() > 0 || initContainer.isNotLoadCompleted()) {
31          try {
32            lastImageImprove = initContainer.getImage(lastImageImprove);
33            if (lastImageImprove != null) {
34              if (!lastImageImprove.isImproved(this)) {
35                System.out.println("Improved image: " + lastImageImprove.getId() + ", thread: "
36                    + Thread.currentThread().getName());
37                increaseImageImprover();
38                lastImageImprove.improve(this);
39                TimeUnit.MILLISECONDS.sleep(100);
40                totalImagesImprovedByThread++;
41              }
42            }
43          } catch (NullPointerException | InterruptedException e) {
44            e.printStackTrace();
45          }
46        }
47      }
48
49      public String getName() {
50        return name;
51      }
52
53      public int getTotalThreadsImprovements() {
54        return totalThreadsImprovements;
55      }
56
```

```
57        public int getTotalImagesImprovedByThread() {
58          return totalImagesImprovedByThread;
59        }
60
61        public void increaseImageImprover() {
62          totalImprovements++;
63        }
64      }
65
```

Código 7: Clase InitContainer

```
1     package TP1;
2
3     import java.util.Random;
4
5     public class InitContainer extends Container {
6       private boolean loadCompleted;
7       private int targetAmountOfImages; // cantidad Max de imagenes a crear
8       private int amountOfImages; // cantidad de imagenes actual
9
10      // llaves
11      private static final Object keyLoad = new Object();
12      private static final Object keyCloneDelete = new Object();
13
14      public InitContainer(int targetAmountOfImages) {
15        loadCompleted = false;
16        this.targetAmountOfImages = targetAmountOfImages;
17        this.amountOfImages = 0;
18      }
19
20      public boolean load(Image image, Loader loader, int cantidad) throws Exception {
21        synchronized (keyLoad) {
22          if (!loadCompleted) {
23            container.addLast(image);
24            amountOfImages++;
25            System.out.println("Loaded image: " + image.getId());
26            if (amountOfImages == targetAmountOfImages) {
27              loadCompleted = true;
28              loader.setImageLoad(cantidad + 1);
29              throw new Exception("Contenedor lleno");
30            }
31          }
32          return loadCompleted;
33        }
34      }
35
36      public Image getImage(Image last) {
37        if (container.size() > 0) {
38          int aux = new Random().nextInt(container.size());
39          return container.get(aux);
```

```
40          } else {
41            return null;
42          }
43        }
44
45        public boolean isNotLoadCompleted() {
46          return !loadCompleted;
47        }
48
49        public Image CopyAndDeleted(Image image) throws InterruptedException {
50          synchronized (keyCloneDelete) {
51            if (container.size() > 0 && image.isIamDeletefromInitContainer()) {
52              Image forClone = new Image(image.getImprovements(), image.getAmIResized(),
    ↪ image.getId(), true,
53                  image.getAmIImproved());
54              this.container.remove(image);
55              //System.out.printf("Imagen copiada y borrada del contenedor inicial: " +
    ↪ forClone.getId());
56              return forClone;
57            } else {
58              return null;
59            }
60          }
61        }
62
63        public int getAmountOfImages() {
64          return amountOfImages;
65        }
66      }
67
```

Código 8: Clase Loader

```
1     package TP1;
2
3     import java.util.concurrent.TimeUnit;
4
5     public class Loader implements Runnable {
6
7       private int imageLoad;
8
9       private final InitContainer initContainer;
10
11      private final String name;
12
13      public Loader(InitContainer initContainer, String name) {
14        this.initContainer = initContainer;
15        this.name = name;
16        imageLoad = 0;
17
18      }
```

```
19
20      @Override
21      public void run() {
22        while (initContainer.isNotLoadCompleted()) {
23          try {
24            if (!initContainer.load(new Image(), this, imageLoad)) {
25              increaseImageLoad();
26              TimeUnit.MILLISECONDS.sleep(50);
27            }
28          } catch (Exception e) {
29            e.printStackTrace();
30            break;
31          }
32        }
33      }
34
35      public void setImageLoad(int imageLoad) {
36        this.imageLoad = imageLoad;
37      }
38
39      public String getName() {
40        return name;
41      }
42
43      public int getImageLoad() {
44        return imageLoad;
45      }
46
47      public void increaseImageLoad() {
48        imageLoad++;
49      }
50
51    }
52
```

Código 9: Clase Log

```
1   package TP1;
2
3   import java.io.FileWriter;
4   import java.io.IOException;
5   import java.io.PrintWriter;
6   import java.util.*;
7   import java.util.concurrent.TimeUnit;
8
9   public class Log extends Thread {
10    private final int targetAmountOfData;
11    private final Date initTime;
12    private final InitContainer initContainer;
13    private final FinalContainer finalContainer;
14
```

```
15      private final Loader[] loaders;
16      private final Improver[] improvers;
17      private final Resizer[] resizers;
18
19      private final Cloner[] cloners;
20
21      private final Thread[] loadersThreads;
22      private final Thread[] improversThreads;
23      private final Thread[] resizersThreads;
24
25      private final Thread[] clonersThreads;
26
27      public static void clearFile() {
28        try {
29          PrintWriter pw_log = new PrintWriter(".//Estadistica.txt");
30          pw_log.print("");
31          pw_log.close();
32        } catch (IOException e) {
33          e.printStackTrace();
34        }
35      }
36
37      public Log(int targetAmountOfData, InitContainer initContainer,
38          FinalContainer finalContainer,
39          Loader[] loaders,
40          Improver[] improvers,
41          Resizer[] resizers,
42          Cloner[] cloners,
43          Thread[] loadersThreads,
44          Thread[] improversThreads,
45          Thread[] resizersThreads,
46          Thread[] clonersThreads) {
47        this.initContainer = initContainer;
48        this.finalContainer = finalContainer;
49        this.improvers = improvers;
50        this.loaders = loaders;
51        this.resizers = resizers;
52        this.cloners = cloners;
53        this.improversThreads = improversThreads;
54        this.resizersThreads = resizersThreads;
55        this.clonersThreads = clonersThreads;
56        this.loadersThreads = loadersThreads;
57        this.targetAmountOfData = targetAmountOfData;
58        initTime = new Date();
59      }
60
61      @Override
62      public void run() {
63        while (finalContainer.getSize() <= targetAmountOfData) {
64          try {
65            writeLog();
```

```
66              TimeUnit.MILLISECONDS.sleep(500);
67            } catch (InterruptedException e) {
68              writeLog();
69              break;
70            }
71          }
72        }
73
74        private void writeLog() {
75          try {
76            PrintWriter pw_log = new PrintWriter(new FileWriter(".//Estadistica.txt", true));
77            pw_log.print("*---------------------------------------------------------------------------*\n");
78            pw_log.printf("Execution time: %.3f [Seg]\n", (float) (new Date().getTime() -
    ↪ initTime.getTime()) / 1000);
79            pw_log.printf("InitContainer size at this moment: %d\n", initContainer.getSize());
80            pw_log.printf("InitContainer size: %d\n", initContainer.getAmountOfImages());
81            pw_log.printf("finalContainer size: %d\n", finalContainer.getSize());
82            pw_log.print("*---------------------------------------------------------------------------*\n\n");
83            Loader[] loadersCopy = loaders;
84            Improver[] improversCopy = improvers;
85            Resizer[] resizersCopy = resizers;
86            Cloner[] clonersCopy = cloners;
87
88            ////////////////// LOADERS ////////////////////////////////
89
90            int totalImageLoad = 0;
91            for (Loader load : loadersCopy) {
92              totalImageLoad += load.getImageLoad();
93            }
94            pw_log.println("   Total loaders:\n");
95            pw_log.printf("     Loaded images: %d\n", totalImageLoad);
96            pw_log.println("");
97            pw_log.printf("     Loaders: \n");
98            for (Loader loader : loadersCopy) {
99              pw_log.printf("        %s:\n", loader.getName());
100             pw_log.printf("          loaded images: %d\n", loader.getImageLoad());
101           }
102           pw_log.println("");
103
104           /**
105           * ////////// IMPROVERS /////////
106           */
107
108           int totalImageImproved = 0;
109           for (Improver improver : improversCopy) {
110             totalImageImproved += improver.getTotalImagesImprovedByThread();
111           }
112           pw_log.println("   Total Improvers:\n");
113           pw_log.printf("     Improved images: %d\n", totalImageImproved);
114           pw_log.println("");
115           pw_log.printf("     Improvers: \n");
```

```
116             for (Improver improver : improversCopy) {
117               pw_log.printf("        %s:\n", improver.getName());
118               pw_log.printf("          improved images: %d\n",
    ↪ improver.getTotalImagesImprovedByThread());
119             }
120             pw_log.println("");
121
122             /**
123             * ////////// RESIZERS //////////
124             */
125
126             int totalImageResized = 0;
127             for (Resizer resizer : resizersCopy) {
128               totalImageResized += resizer.getTotalImagesResized();
129
130             }
131             pw_log.println("   Total Resizers:");
132             pw_log.println("");
133             pw_log.printf("     Resized images: %d\n", totalImageResized);
134             pw_log.println("");
135             pw_log.printf("     Resizers: \n");
136             for (Resizer resizer : resizersCopy) {
137               pw_log.printf("        %s:\n", resizer.getName());
138               pw_log.printf("          resized images: %d\n", resizer.getTotalImagesResized());
139               pw_log.printf("          responsibility percentage in copied data over target data: %.2f
    ↪ %%\n",
140                     100 * (float) resizer.getTotalImagesResized() / totalImageLoad);
141             }
142             pw_log.println("");
143
144             ////////////////// CLONER ////////////////////////////////
145
146             int totalImagesCloned = 0;
147             pw_log.println("   Total Cloners:\n");
148             for (Cloner cloner : clonersCopy) {
149               totalImagesCloned += cloner.getImageCloned();
150             }
151             pw_log.printf("     Cloned images: %d\n", totalImagesCloned);
152             pw_log.println("");
153             pw_log.printf("     Cloners: \n");
154             for (Cloner cloner : clonersCopy) {
155               pw_log.printf("        %s:\n", cloner.getName());
156               pw_log.printf("          cloned images: %d\n", cloner.getImageCloned());
157               pw_log.printf("          responsibility percentage in taken data over target data: %.2f
    ↪ %%\n",
158                     100 * (float) cloner.getImageCloned() / totalImageLoad);
159             }
160             pw_log.println("");
161
162             ////////////////// Estados de los Hilos ////////////////////////////////
163
```

```
164             pw_log.println("   Threads State:\n");
165
166             for (Thread loaderThread : loadersThreads) {
167               pw_log.printf("      %s: %s\n", loaderThread.getName(),
    ↪ loaderThread.getState().name());
168             }
169
170             pw_log.println();
171
172             for (Thread improverThread : improversThreads) {
173               pw_log.printf("      %s: %s\n", improverThread.getName(),
    ↪ improverThread.getState().name());
174             }
175
176             pw_log.println();
177
178             for (Thread resizerThread : resizersThreads) {
179               pw_log.printf("      %s: %s\n", resizerThread.getName(),
    ↪ resizerThread.getState().name());
180             }
181
182             pw_log.println();
183
184             for (Thread clonerThread : clonersThreads) {
185               pw_log.printf("      %s: %s\n", clonerThread.getName(),
    ↪ clonerThread.getState().name());
186             }
187
188             pw_log.println();
189             pw_log.print("*------------------------------------------------------------------------*\n\n");
190             pw_log.close();
191
192           } catch (IOException e) {
193             e.printStackTrace();
194           }
195         }
196       }
197
```

Código 10: Clase Resizer

```
1   package TP1;
2
3   import java.util.concurrent.TimeUnit;
4
5   public class Resizer implements Runnable {
6
7     private final InitContainer initContainer;
8
9     private final String name;
10
```

```
11      private Image lastImageResized;
12
13      private int totalImagesResized;
14
15      public Resizer(InitContainer initContainer, String name) {
16        this.initContainer = initContainer;
17        this.name = name;
18        lastImageResized = null;
19        totalImagesResized = 0;
20        lastImageResized = null;
21      }
22
23      public int getTotalImagesResized() {
24        return totalImagesResized;
25      }
26
27      @Override
28      public void run() {
29        while (initContainer.getSize() > 0 || initContainer.isNotLoadCompleted()) {
30          try {
31            lastImageResized = initContainer.getImage(lastImageResized);
32
33            if (lastImageResized != null) {
34
35              if (lastImageResized.getAmIImproved()) {
36
37                if (lastImageResized.resize()) {
38                  System.out.println("Image : " + lastImageResized.getId() + " by
39                  thread: "
40                  + Thread.currentThread().getName());
41                  increaseImageResizer();
42                  TimeUnit.MILLISECONDS.sleep(100);
43                }
44              }
45            }
46          } catch (NullPointerException e) {
47            e.printStackTrace();
48            System.out.println("SE BORRO DATO DEL CONTAINER, INTENTO DE NUEVO");
49          } catch (InterruptedException | IndexOutOfBoundsException e) {
50            e.printStackTrace();
51          } catch (IllegalArgumentException e) {
52            e.printStackTrace();
53            break;
54          }
55        }
56      }
57
58      public InitContainer getInitContainer() {
59        return initContainer;
60      }
61
```

```
62        public String getName() {
63          return name;
64        }
65
66        public void increaseImageResizer() {
67          totalImagesResized++;
68        }
69
70        public Image getlastImageResized() {
71          return lastImageResized;
72        }
73
74      }
75
```

Las clases anteriormente descritas resultan en un diagrama de clases UML como se ve a continuación:

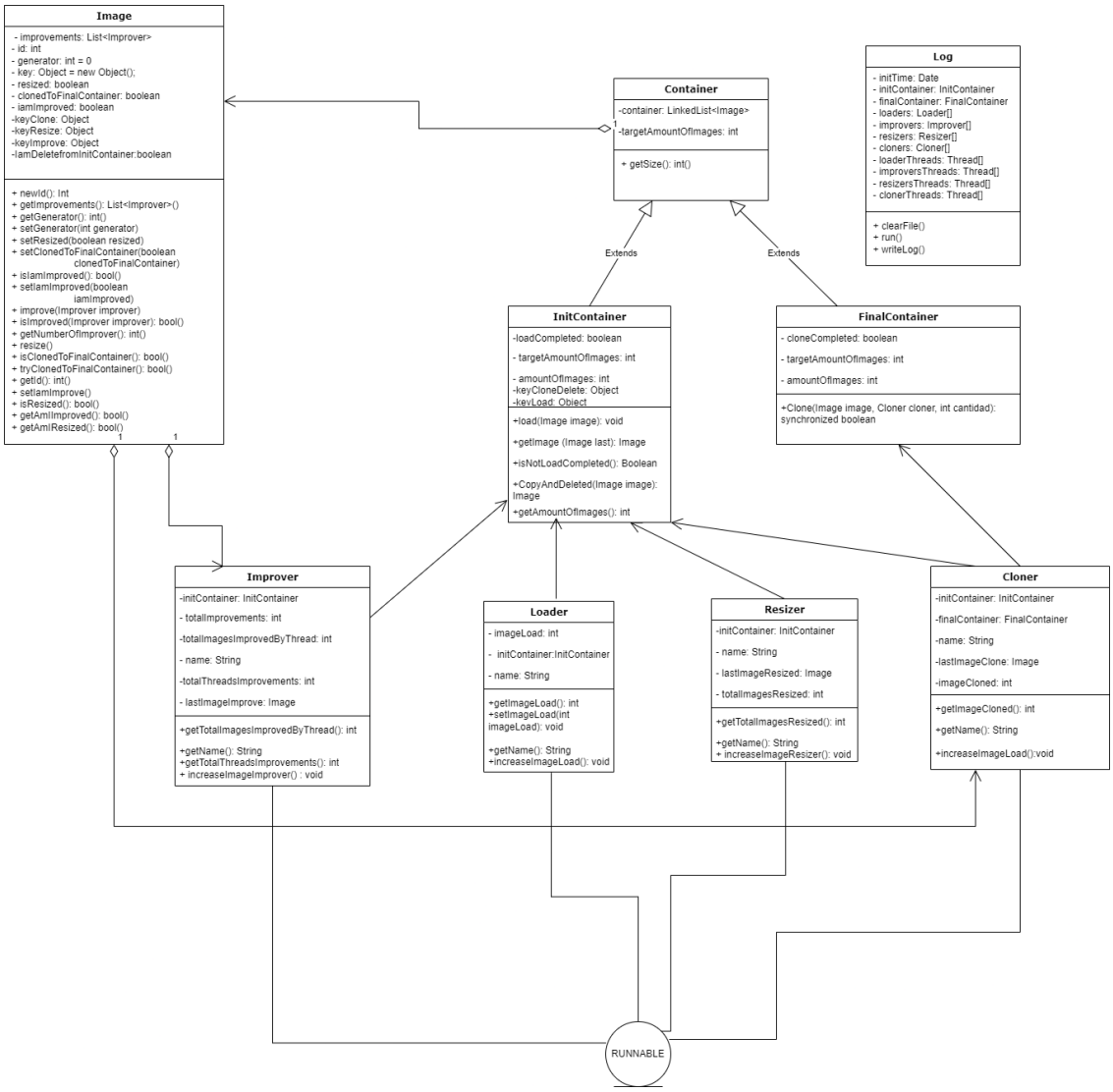

Figura 1: Diagrama de Clases

Y, el funcionamiento del código puede apreciarse en el siguiente diagrama de secuencia.

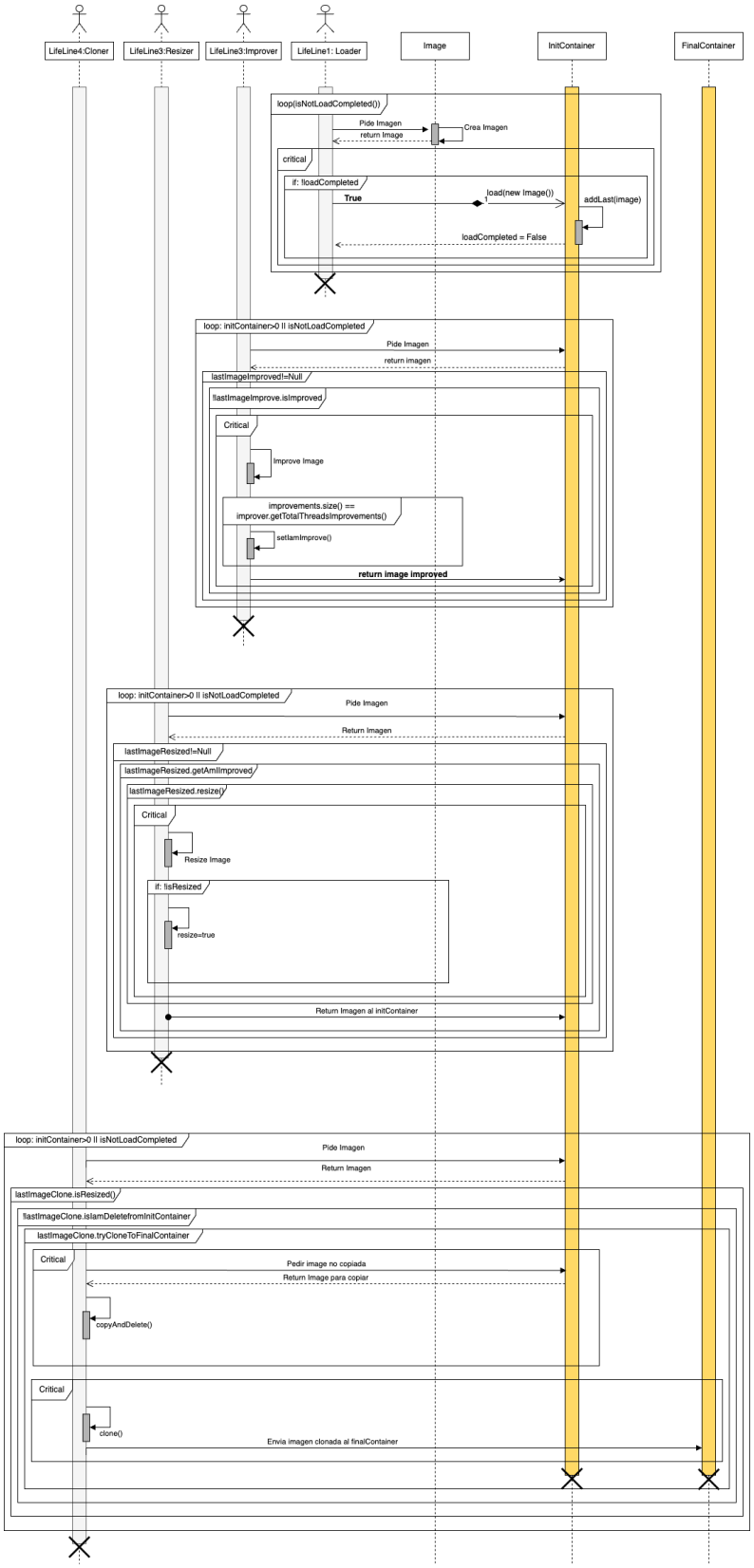

Figura 2: Diagrama de Secuencia

2.1.3. Estructura del código

A continuación se detallan los métodos principales de cada una de las clases implementadas:

- **Image**: Constructor de la clase. Recibe como parámetros: **List<Improver> improvements**, un **boolean resized**, **int id**, **boolean clonedToFinalContainer**, **boolean iAmImproved**.
  - **improve(Improver improver)**: método que realiza la acción de mejorar la imagen.
  - **isIamImproved()**: método que retorna true si la imagen ya fue mejorada por los tres improvers.
  - **isIamDeletefromInitContainer()**: método booleano que retorna **true** si la imagen fue eliminada del initContainer.
  - **isImproved(Improver improver)**: (bool), dice si un hilo del proceso Improver mejoró o no la imagen.
  - **resize()**: este método devuelve **true** si la imagen no fue reescalada por ningún hilo.
  - **isClonedToFinalContainer()**: (bool), este método retorna **true** si la imagen fue clonada en el contenedor final.
  - **tryClonedToFinalContainer()**: (bool), retorna **true** si la imagen no fue eliminada de initContainer.
  - **isResized()**: (bool), indica si la imagen fue reescalada o no.
- **Container**: esta clase representa el espacio de almacén de los datos.
  - **initContainer**: esta clase extiende de la clase **Container**. Aquí se cargan las imágenes iniciales.
    - **load(Image image)**: (void), carga una imagen en el contenedor.
    - **isNotLoadCompleted()**: (boolean), retorna **true** si la carga no está completa.
    - **CopyAndDeleted(Image image)**: (Image), copia la imagen y la borra del initContainer.
  - **FinalContainer**: extiende de **Container**. Aquí se guardarán las imágenes ya mejoradas, reescaladas y copiadas.
    - **FinalContainer()**: constructor de la clase, que recibe un parámetro del tipo int con el número máximo de imágenes a trabajar.
    - **Clone(Image image, Cloner cloner, int cantidad)**: (synchronized boolean), este método consulta si el clonado está completo, siempre y cuando esté completo, la imagen no sea nula y la imagen haya sido recortada, agrega la imagen al FinalContainer. En caso de que la imagen sea la última, se detiene el trabajo de los Cloners.
- **Loader**: clase del proceso encargado de cargar las imágenes.
  - **Loader()**: constructor de la clase, recibe como parámetros un **InitContainer** y un String con el nombre de cada hilo.
  - **imageLoad()**: (void), carga una imagen.
  - **increaseImage()**: aumenta en uno un contador.

  - **Resizer**: clase del proceso que se encarga de reescalar las imágenes.
    - **Resizer()**: constructor de la clase, recibe un parámetro de tipo **InitContainer** y un String con el nombre.
    - **increaseImageResized()**: (void), incrementa en uno la cantidad de imágenes reescaladas (**totalImagesResized++**).

- **Cloner**: clase del proceso encargado de clonar las imágenes del contenedor inicial.
  - **Cloner()**: constructor de la clase, recibe como parámetro un **InitContainer**, un **FinalContainer** y un string con el nombre del hilo.
  - **increaseImageClone()**: (void), método que incrementa en uno la cantidad de imágenes ya clonadas.

2.1.4. Herramientas utilizadas

Para escribir el código fuente del proyecto de utilizaron las siguientes herramientas:

- **IntelliJ IDEA Ultimate 2022.3**: para escribir el código y realizar las pruebas necesarias.
- **GitHub**: para mejorar la colaboración entre los participantes y poder mantener un control de versiones.
- **Diagrams.net**: para la realización de los diagramas UML requeridos.

3. Resultados obtenidos

3.1. Aspectos generales

Para la obtención de los tiempos de ejecución pedidos, se trabajó con diferentes tiempos de sleep dentro de las siguientes clases:

- Loader
- Improver
- Resizer
- Cloner

La salida del log **Estadisticas.txt** tiene el siguiente formato:

Código 11: Estadistica.txt

```
*---------------------------------------------------------------------------*
Execution time: 10.446 [Seg]
InitContainer size at this moment: 0
InitContainer size: 100
finalContainer size: 100
*---------------------------------------------------------------------------*

Total loaders:

   Loaded images: 100

   Loaders:
      TP1.Loader 0:
         loaded images: 50
      TP1.Loader 1:
         loaded images: 50

Total Improvers:

   Improved images: 300

   Improvers:
      TP1.Improver 0:
         improved images: 100
      TP1.Improver 1:
         improved images: 100
      TP1.Improver 2:
         improved images: 100

Total Resizers:

   Resized images: 100

```

```
        Resizers:
            TP1.Resizer 0:
                resized images: 34
                responsibility percentage in copied data over target data: 34.00 %
            TP1.Resizer 1:
                resized images: 32
                responsibility percentage in copied data over target data: 32.00 %
            TP1.Resizer 2:
                resized images: 34
                responsibility percentage in copied data over target data: 34.00 %

    Total Cloners:

        Cloned images: 100

        Cloners:
            TP1.Cloner 0:
                cloned images: 51
                responsibility percentage in taken data over target data: 51.00 %
            TP1.Cloner 1:
                cloned images: 49
                responsibility percentage in taken data over target data: 49.00 %

    Threads State:

        TP1.Loader 0 (Thread ID: 12): TERMINATED
        TP1.Loader 1 (Thread ID: 13): TERMINATED

        TP1.Improver 0 (Thread ID: 14): TERMINATED
        TP1.Improver 1 (Thread ID: 15): TERMINATED
        TP1.Improver 2 (Thread ID: 16): TERMINATED

        TP1.Resizer 0 (Thread ID: 17): TERMINATED
        TP1.Resizer 1 (Thread ID: 18): TERMINATED
        TP1.Resizer 2 (Thread ID: 19): TERMINATED

        TP1.Cloner 0 (Thread ID: 20): TERMINATED
        TP1.Cloner 1 (Thread ID: 21): TERMINATED

    *------------------------------------------------------------------------*

```

3.2. Observaciones

Para la elección de los tiempos aleatorios, el grupo hizo pruebas estadísticas, primero, usando tiempos aleatorios iguales en cada uno de los procesos, se obtiene así entonces:

- 1ra iteración - 10ma iteración:
  - Loaders: 50 ms

  - Improvers: 50 ms
  - Improvers: 50 ms
  - Resizers: 50 ms
  - Cloners: 50 ms

Luego de hacer un promedio de los datos se llegó a a conclusión de que el límite inferior de tiempo (en promedio) es aproximadamente de 11.7 segundos:

- 11va iteración - 20va interación:
  - Loaders: 100 ms
  - Improvers: 100 ms
  - Resizers: 100 ms
  - Cloners: 100 ms

Luego, para las siguientes iteraciones se decidió modificar los valores de los hilos individualmente. Primero se modificó el tiempo de los Loaders, luego, el de los Improvers, tercero el de los Resizers, y por último, los cloners. De esto último, se llega a la conclusión de que cuando se cambiaban los tiempos de los improvers y los resizers, se alteraba sustancialmente os valores del tiempo de ejecución final.

Luego de todas estas pruebas, se estableció un rango de valores, entre 50 ms como mínimo y 100 ms como máximo, y al ser los valores más importantes los de los improvers y los resizers, se decidió:

- Loaders: 50 ms
- Improvers: 100 ms
- Resizers: 100 ms
- Cloners: 50 ms

Estas ejecuciones tardan en promedio (y después de 100 ejecuciones probadas) unos 10.5 segundos como mínimo y 13.5 segundos como máximo, cumpliendo de esta forma los límites establecidos en la consigna del trabajo práctico.

4. Conclusiones

Al final de la realización de este trabajo práctico, se pueden extraer las siguientes conclusiones:

- Se entendió la importancia de manejar de forma correcta la sincronización de los hilos, ya que, de lo contrario, si no son debidamente administradas las zonas en donde existe acceso a la memoria compartida (secciones críticas), se compromete la integridad de los datos y puede producir resultados inesperados.
- Se pudo determinar la mejor herramienta de manejo para la sincronización de los hilos, pudiendo así entender las diferencias existentes entre unas y otras. Para el caso del trabajo práctico, se utilizó **synchronized**. Se determinó que era la herramienta más adecuada para este contexto.
- Se encontró una relación aproximadamente lineal entre el tiempo de ejecución del programa y la cantidad de imágenes a procesar, de forma que es posible estimar el tiempo de ejecución para el programa. Se podría decir que se tiene una recta de ecuación

  $$t(n) = 0.1n$$

  En donde t representa el tiempo de ejecución, y n es la cantidad de imágenes a procesar.
- Durante el proceso de desarrollo fue posible observar la complejidad del manejo de las variables que intervienen en un proceso concurrente, y, que a medida que crece el programa se hace más dificil no cometer errores.
- Así mismo, fue posible comprobar que en el desarrollo de un programa, el trabajo en equipo en tiempo real agiliza ampliamente el el proceso de codificación.